

# **SANDIA REPORT**

SAND2015-0928

Printed [Month] 2015

## **General Purpose Graphics Processing Unit Based High-Rate Rice Decompression and Reed-Solomon Decoding**

Thomas A. Loughry

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from

U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd.  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online order: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# **General Purpose Graphics Processing Unit- Based High-Rate Rice Decompression and Reed-Solomon Decoding**

Thomas A. Loughry  
Decision Support Systems  
Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, New Mexico 87185-MS0576

## **Abstract**

As the volume of data acquired by space-based sensors increases, mission data compression/decompression and forward error correction code processing performance must likewise scale. This competency development effort was explored using the General Purpose Graphics Processing Unit (GPGPU) to accomplish high-rate Rice Decompression and high-rate Reed-Solomon (RS) decoding at the satellite mission ground station. Each algorithm was implemented and benchmarked on a single GPGPU. Distributed processing across one to four GPGPUs was also investigated. The results show that the GPGPU has considerable potential for performing satellite communication Data Signal Processing, with three times or better performance improvements and up to ten times reduction in cost over custom hardware, at least in the case of Rice Decompression and Reed-Solomon Decoding.

## **ACKNOWLEDGMENTS**

We would like to acknowledge John Feddema, Manager 05521, for his oversight and support in securing funding for this competency development, as well as Todd Jenkins of 05522 for his IT support in setting up the GPGPU/Server configurations.

DRAFT

## CONTENTS

Acknowledgments.....	4
1. Overview.....	9
2. Test Platforms .....	11
3. General Purpose Graphics Processing Unit Programming Model.....	15
4. Rice Compression/Decompression .....	19
4.1. General Purpose Graphics Processing Unit Rice Decompression Algorithm .....	21
4.2. Rice Decompression Software, Central Processing Unit vs. General Purpose Graphics Processing Unit .....	23
4.3. Timing Analysis of the Basic Compute Unified Device Architecture Rice Decompression Algorithm .....	25
4.4. Multiple Host Threads for Rice Decompression .....	26
4.5. Multiple Host Threads, Multiple Devices for Rice Decompression .....	28
5. Reed-Solomon.....	33
5.1. Reed-Solomon Decoding Software vs. General Purpose Graphics Processing Unit.....	34
5.2. Timing Analysis of the Basic CUDA RS Decoding Algorithm .....	35
5.3. Multiple Host Threads, Multiple Devices for Reed-Solomon Decoding .....	36
6. Conclusion .....	39
7. References.....	41

## FIGURES

Figure 1. Tesla S1070 .....	11
Figure 2. Tesla Configuration.....	11
Figure 3. Intel 7300 Chip Set.....	12
Figure 4. Workstation Overview .....	13
Figure 5. Host/Device Interface.....	16
Figure 6. Rice Compression Algorithm.....	20
Figure 7. Kernel 1 – Decompress Data.....	22
Figure 8. Kernel 2 – Remove Predictor and Map Data .....	22
Figure 9. Kernel 3 – CRC Calculation.....	22
Figure 10. Software vs. GPGPU Performance .....	23
Figure 11. Latency .....	25

Figure 12. Timing Analysis .....	26
Figure 13. Multiple Host Threads (Performance).....	27
Figure 14. Multiple Threads (Latency).....	28
Figure 15. Multiple Devices and Threads Performance .....	30
Figure 16. Latency for Multiple Devices and Threads .....	30
Figure 17. Workstation Results .....	31
Figure 18. Transfer Frame Format.....	33
Figure 19. RS Decoding Performance .....	34
Figure 20. RS Decoding Latency.....	35
Figure 21. RS Time Analysis.....	36
Figure 22. Multiple Threads, Multiple Devices RS Decoding .....	36
Figure 23. Multiple Host Thread, Multiple Device RS Latency .....	37
Figure 24. Workstation Performance.....	38

## TABLES

Table 1. S1070/DL580G5 Goals vs. Results .....	10
Table 2. DMA and Compute Bandwidth as a Function of Host Threads and Devices .....	29

## ACRONYMS

ASIC	Application Specific Integrated Circuit
ASM	Asynchronous Symbol Marker
CCSDS	Consultative Committee for Space Data Systems
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CUDA	Compute Unified Device Architecture
DAS	Data Acquisition Subsystem
DMA	Direct Memory Access
DOE	Department of Energy
EP	External Predictor
FPA	Focal Plane Array
FPGA	Field-Programmable Gate Arrays
FSB	Front-Side Bus
GB	Gigabyte
GB/s	Gigabytes per second
GPGPU	General Purpose Graphics Processing Unit
HIC	Host Interface Card
I/O	Input/Output
IOH	Input/Output Hub
MB/s	Megabytes per second
MCH	Memory Controller Hub
MGS	Mission Ground Station
NN	Nearest Neighbor
PCIe	Peripheral Component Interconnect express
PF	Previous Frame
QPI	Quick Path Interconnect
RS	Reed-Solomon
TF	Transfer Frame
SNL	Sandia National Laboratories
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

DRAFT



## 1. OVERVIEW

Using the General Purpose Graphics Processing Unit (GPGPU) to accomplish high-rate Rice Decompression and high-rate Reed-Solomon (RS) decoding was explored. Rice Decompression and RS fall into a class of algorithms that are currently used in satellite ground communication systems implemented at Sandia National Laboratories (SNL). Neither algorithm is inherently parallelizable. Common Rice decoder implementations used at SNL include Application Specific Integrated Circuits (ASICs) capable of data processing rates up to 80 MB/s and software solutions capable of data rates of up to 150 MB/s running on multiple Central Processing Units (CPUs)/cores. Common RS decoding implementations at SNL are primarily accomplished in VHDL (VHSIC [Very High-Speed Integrated Circuit] Hardware Description Language) code in Field-Programmable Gate Arrays (FPGAs) at about 62 MB/s. Although the general wisdom is that algorithms heavy in floating point calculations benefit the most from GPGPU implementations, we hoped to gain significant improvements in performance for both the Rice and RS algorithms using the GPGPU, even though neither algorithm uses any floating point arithmetic.

We investigated using both a single GPGPU device implementation and a distributed implementation with up to four GPGPU devices for each algorithm. In addition, we also varied the number of host threads used to drive the GPGPUs to exploit overlapped GPGPU execution and Input/Output (I/O). Performance goals were defined to provide metrics and to keep the effort focused. When a given goal was achieved, no further refining of the algorithm or implementation was pursued. Goals were chosen for a single GPGPU solution that reflects nearly a doubling of current capabilities as described above. For four GPGPU devices, we scaled the goals by a factor of approximately three over the single device implementation.

As seen in [Table 1](#), we exceeded our expectations for Rice Decompression by a factor of three for the single device implementation and by nearly 38% for the four-device distributed implementation. Also note that for Rice Decompression, no further performance gain was achieved when distributing across more than two devices and in effect two, three, and four device configurations netted the same performance results. This was determined to be the result of saturating the Peripheral Component Interconnect express (PCIe) bus that conveys data between the host computer and the GPGPU. For RS, we also exceeded our goals by a small to moderate margin.

**Table 1. S1070/DL580G5 Goals vs. Results**

Goal Result	Devices	Rice Decmp	R/S
	1	300 MB/s	120 MB/s
	4	1000 MB/s	400 MB/s
	1	900 MB/s	125 MB/s
	4	1377* MB/s	495 MB/s

\* Two Devices

The benchmark results shown in [Table 1](#) were measured on a Tesla S1070, multi-device GPGPU, coupled with an HP DL580G5 server. We also ran both algorithms on two different workstations employing two different, inexpensive video graphics cards. Specifically, we looked at the NVidia GTX 280 which employs the same GPGPU chip as used by the S1070; and a GTX 480 based on NVidia's next generation Fermi Technology. The GTX 280 performed slightly better than the S1070 for single GPGPU, single host thread operation. The slight increase in performance can be accounted for as a result of the workstation's improved PCIe performance over the DL580G5. The GTX 480 performed twice as well as the GTX 280 and S1070 for single device and single host thread operation. We did, however, find limitations in both video graphics cards due to driver performance when using more than one host thread.

## 2. TEST PLATFORMS

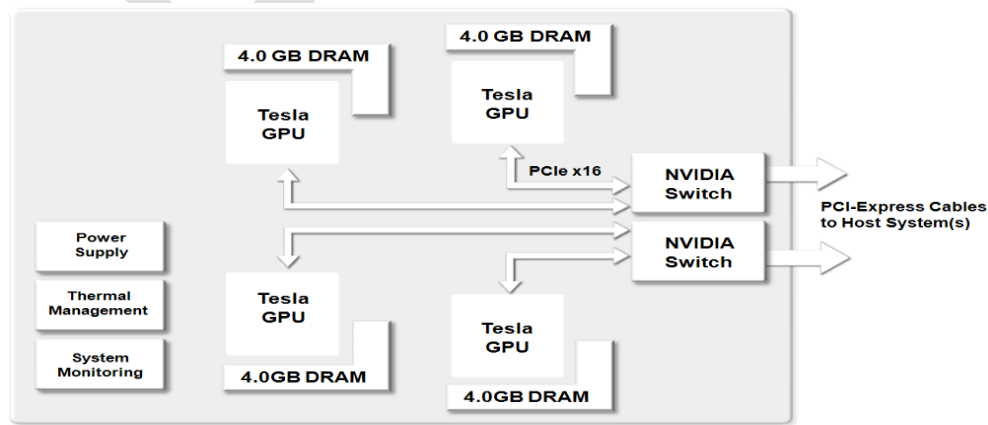
The test platform for this competency development effort primarily consisted of a Tesla S1070 and a HP DL580G5 server. We also ran benchmarks on a Core 2 Duo-based workstation with a GTX 280 graphics card installed and a Core i7-based workstation with a GTX 480 graphics card installed.

The Tesla S1070 (Figure 1) is a 1U rack-mount chassis containing four GPGPUs, control unit, and power supply. Information on the S1070 is available on NVidia's web site at [www.nvidia.com](http://www.nvidia.com). Each GPGPU has 4 GB of dedicated GDDR3 memory with a 512-bit interface and can achieve memory access bandwidths to its dedicated memory at rates as high as 102 GB/s. The S1070 interfaces to the host computer using two Host Interface Cards (HICs). Each HIC uses a PCIe switch to access its two connected GPGPUs.



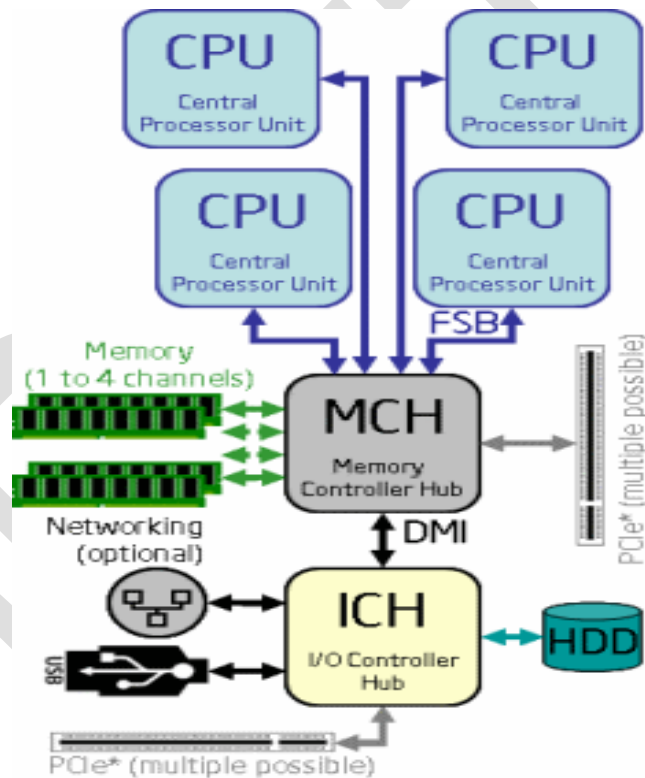
**Figure 1. Tesla S1070**

The S1070 allows the two HIC cards to be placed in two completely different hosts, if desired, as shown in Figure 2. Although the S1070 HICs support PCIe Gen 2.0 at x16 lanes each, our server limited us to PCIe Gen 1 at x8 lanes each. This effectively limited the bandwidth between the host and GPGPU to less than 2 GB/s in the aggregate. All of our GPGPU code was developed in NVidia's Compute Unified Device Architecture (CUDA) language, version 3.2.



**Figure 2. Tesla Configuration**

The HP DL580G5 is a high-end server with four Xeon X7460 CPUs running at 2.66 GHz each. Each CPU contains six cores for a total of 24 cores. The CPUs interface to the rest of the system through Intel's 7300 series chipset (Figure 3). More information on the chip set can be found at [www.intel.com](http://www.intel.com). As configured for our tests, the DL580G5 has 16 GB of PC2-5300 DDR2 memory with a theoretical bandwidth of 21.2 GB/s read and 10.7 GB/s write speeds. As mentioned previously, the DL580G5 only supports 8 electrical Gen 1 PCIe lanes. It also appears that the physical lanes may be switched, limiting the aggregate bandwidth on the PCIe bus to less than 2.0 GB/s. This is investigated further in Section 4.5.



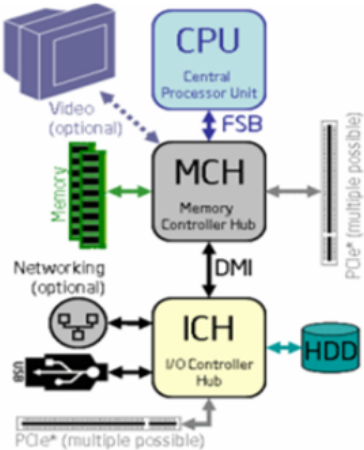
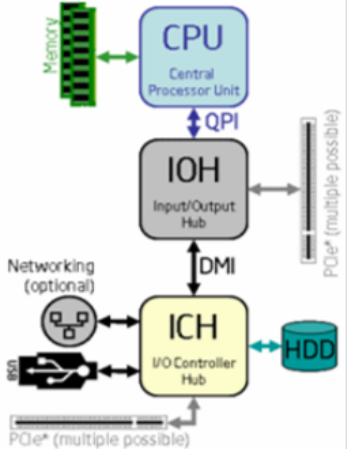
**Figure 3. Intel 7300 Chip Set**

The DL580G5 was selected out of convenience, not for its CPU performance. In practice, because the GPGPU is burdened with most of the computations associated with the two algorithms, a much lower-end workstation with fewer CPUs and better PCIe performance would be a better choice. The DL580G5 was running SLSES10 SP2 and the host code was compiled using g++. The GPGPU code was compiled using CUDA version 3.2.

We also benchmarked on two different workstations that employ commodity graphics cards and Microsoft's Windows 7 OS (Figure 4). The first system, TAL-2000, is based on a Q6600, 4-core CPU running at 3.0 GHz and an Intel X38 chipset. It was configured with 8GB of PC2-5300 DDR2 memory with a theoretical limit of 10.6 GB/s. The mother board supports a full 16 lane

Gen 2 PCIe slot for the video graphics card (GPGPU). In theory, the PCIe slot should support 8.0 GB/s, but was measured at 3.1 and 2.6 GB/s read and write speeds respectively with the GTX 280 installed. NVidia reports that the GTX 280 is capable of 5.2 GB/s data transfer across the bus.

The second workstation, TAL-4000, was based on an Intel Core i7 960, 4-core CPU running at 3.2 GHz and using the Intel X58 chipset. It was configured with 6 GB of PC3-12800 DDR3 memory with a theoretical bandwidth of 38.4 GB/s and a GTX 480 graphics card in a full 16 lane Gen 2 PCIe slot. However, the Quick Path Interconnect (QPI) limits the Direct Memory Access (DMA) bandwidth to 12.8 GB/s between the CPU and the Input Output Hub (IOH). The measured bandwidth to the GTX480 card on this workstation was measured at 5.2 and 5.9 GB/s read and write speed respectively. Although the Windows 7 driver for these two cards support CUDA, it is optimized for game play and does not support the entire feature set of the GPGPU compute capability, including simultaneous data transfer and computes.

Architecture	TAL-2000	TAL-4000
		
CPU	Q6600 (4 cores)	I7 960 (4 cores)
CPU Clock	3.0 GHz O.C.	3.2 GHz
North Bridge (NB)	X38	X58
CPU/NB Interface	FSB 333 MHz O.C. 10.6 GB/s	QPI 6.4 GT O.C. 12.8 GB/s
Memory	8GB DDR2, PC2 5300 10.6 GB/s	6GB DDR3, PC3 12800 38.4 GB/s
Graphics Card	GTX280	GTX480
GPU Interface	PCIe G2 x16 8 GB/s	PCIe G2 x16 8 GB/s
Measured GPU/Host Bandwidth	H2D: 2.6 GB/s D2H: 3.1 GB/s	H2D: 5.2 GB/s D2H: 5.9 GB/s

#### **Figure 4. Workstation Overview**

The Tesla S1060 and the GTX 280 graphics card use the same basic GPGPU design. Each contains 240 scalar processors and 30 multi-processors. The GTX 480 is based on NVidia's newer GPU which was released mid-year 2010. The GTX 480 has 480 scalar processors and 15 multi-processors. Because of design changes in the multi-processors, the GTX 480 has roughly twice the performance of the GTX 280, even though it has half the multi-processors.

DRAFT

### 3. GENERAL PURPOSE GRAPHICS PROCESSING UNIT PROGRAMMING MODEL

For Rice Decompression, parallelism is achieved by collecting many compressed packets and then decompressing the entire set nearly simultaneously on the GPGPU. Likewise, for Reed-Solomon decoding, we collect many code blocks and decode them all at once. In modern satellite communication systems, data continuously flows; hence, while one block of data is being decoded by the GPGPU, another block of data can be accumulated by the host. In the remainder of this document, the block of data being accumulated or processed is referred to as a buffer. Clearly in both of these cases, Rice and RS, GPGPU device thread divergence is a concern, so although we did not expect spectacular performance improvements over the CPU model, we did expect moderate improvements.

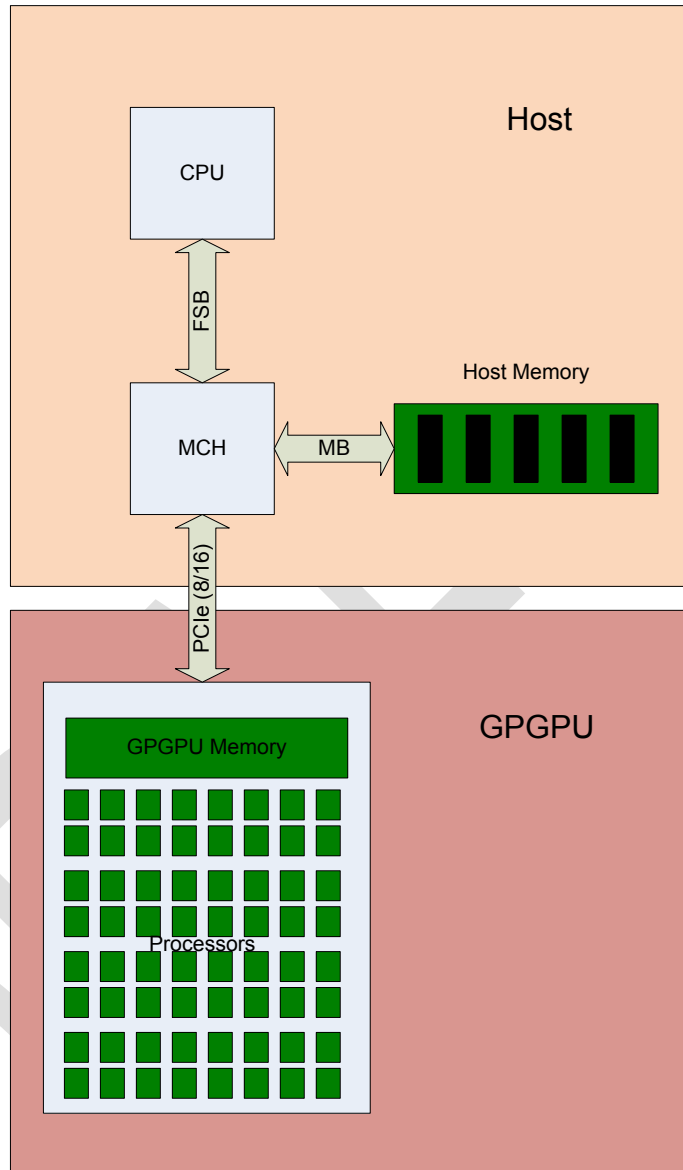
In general, using a GPGPU for data processing is a five-step process:

**Step 1.** Select the device (when multiple devices exist on the same host), allocate device memory, copy constant values to the device, and set the device behavioral flags. We commonly refer to the GPGPU as the “device” and the host computer as the “host.” Generally, this step is only performed once and is designed to prepare the device for computing.

**Step 2.** Transfer the block of data (buffer) to be operated on, Rice compressed packets or RS code blocks, from host memory to the GPGPU memory. As depicted in Figure 5, for Front-Side Bus (FSB) based systems such as the DL580G5 and the TAL-2000, this usually involves DMA through the Memory Controller Hub (MCH), commonly referred to as the “north bridge.” For QPI-based systems such as the TAL-4000, the memory controller is hosted in the CPU and thus DMA occurs through the QPI bus and the IOH, as shown in Figure 4. The QPI bus is typically much faster than the FSB.

**Step 3.** Launch one or more kernels sequentially on the device. Each kernel computes part of the algorithm using one thread per compressed packet or code block. The GPGPU is capable of having thousands of threads in play at once. Thus, thousands of packets or code blocks can be operated on in a near simultaneous fashion.

**Step 4.** This step is very similar to Step 2, except now the results are transferred from the device back into the host memory. In the case of Rice Decompression, if one assumes a 3:1 compression ratio, the size of the data being moved in this step is three times larger than the size of the data moved in Step 2. In contrast, for RS, the data moved in this step is small compared to the input data size because only the corrected symbols and status need to be transferred back to the host. It is also worth mentioning that an alternative to using DMA for memory transfers is a method called “zero copy.” Zero copy uses mapped memory as opposed to DMA. Zero copy can be useful when the size of the data to be copied from host to device or device to host is relatively small.



**Figure 5. Host/Device Interface**

**Step 5.** This last step is simply a branch back to Step 2. While Steps 2 through 4 are being performed, the host computer is accumulating the next set of data (buffer) on which to operate.

In practice, Mission Ground Station (MGS) data processing is pipelined within the same host server and across multiple host servers and custom hardware. Pipeline processing achieves concurrency through functional decomposition with the different functions processing on separate CPU threads or hardware. Within the same stage of a processing pipeline, data decomposition can be used to achieve further concurrency. For example, if data is down-linked on multiple independent channels, each link can be processed separately by different threads or hardware in the RS stage. Likewise, if a satellite possesses multiple Focal Plane Arrays (FPAs),



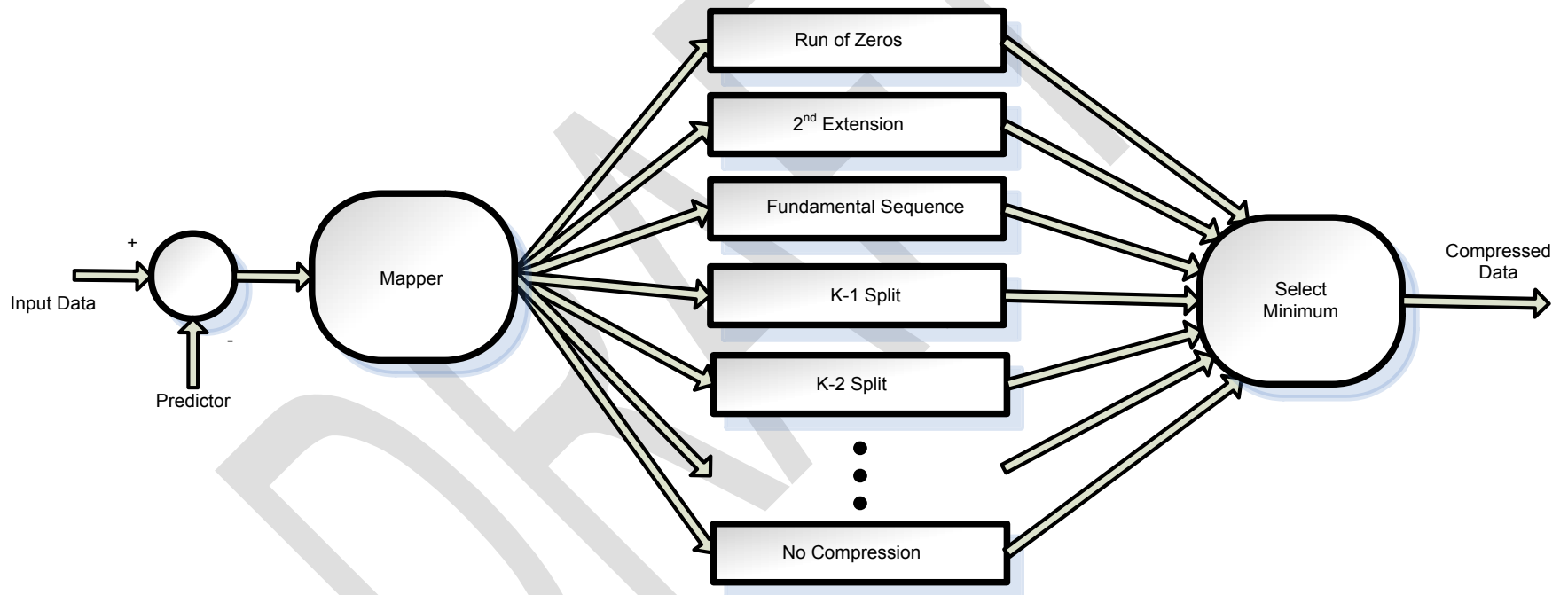
the data from each FPA can be decompressed separately using independent threads or devices. This concept of functional and data decomposition can easily be implemented on multiple GPGPU devices like the S1070 described above.

DRAFT

DRAFT

## 4. RICE COMPRESSION/DECOMPRESSION

Rice compression is an adaptive, noiseless and lossless compression technique developed by Robert F. Rice of the Jet Propulsion Laboratory, National Aeronautics and Space Administration. Rice compression is also recommended by the Consultative Committee for Space Data Systems (CCSDS). A detailed description of Rice compression/decompression can be found in the *CCSDS Green Book 121.0-B-2* [Ref 1]. As depicted in Figure 6, the first step in compressing an image packet is to decorrelate the pixel data and map it into sigma values for entropy encoding. In this effort, we looked at data that was compressed using two common decorrelation techniques: unit delay decorrelation, commonly referred to as Nearest Neighbor (NN); and External Predictor (EP) decorrelation, commonly referred to as Previous Frame (PF).



**Figure 6. Rice Compression Algorithm**

Post decorrelation, the sigma values are compressed using one of 16 different approaches and the approach that achieves the greatest compression is selected as the result. In our application, the data is compressed 16 pixels at a time. An uncompressed packet always contains exactly 4,096 15-bit pixels that correspond to a 64X64 pixel image. Therefore, each packet results in 256 compressed, variable sized blocks that are concatenated to form the compressed packet. Each compressed block starts with a 4-bit ID that identifies the compression method selected by the entropy encoder to compress that 16 pixel block.

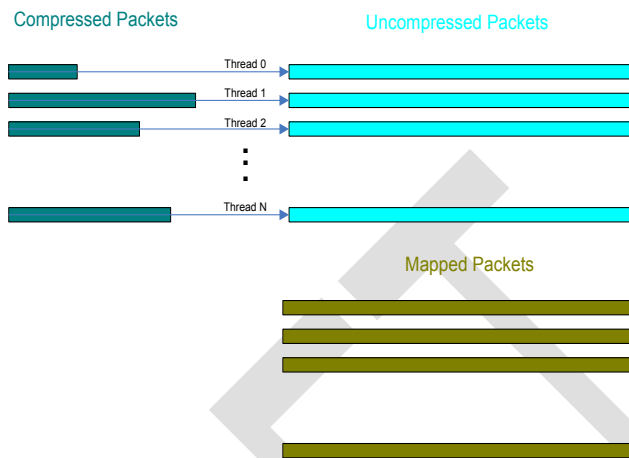
Decompression cannot be performed in parallel on a single packet because there is no way of knowing a priori where the 4-bit ID of each block, other than the first, falls in the bit stream. Instead, the blocks must be decompressed one at a time and in order, so that the ID of the next block can be found in the bit pattern.

#### **4.1. General Purpose Graphics Processing Unit Rice Decompression Algorithm**

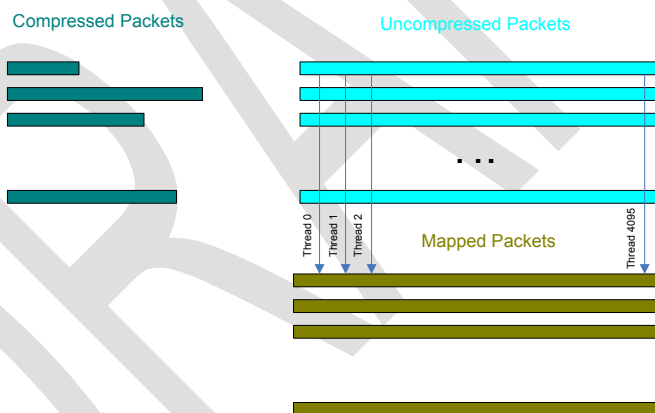
As stated previously, in a real-time MGS, image data will flow continuously, providing an opportunity for many compressed packets to be accumulated in a very short period of time. It should also be noted that compressed packets generally include a Cyclic Redundancy Check (CRC) at the end of the packet body that was calculated prior to compression. The CRC provides a quick check of the integrity of the resulting packet after decompression, i.e., the calculated CRC of the resulting uncompressed packet should match the original CRC calculated before compression and down-linking. In our current ground systems, we use both hardware and software for decompression. We also calculate the CRC and do some simple mappings of the data (such as rotations and flips) in the same decompression hardware or software module to exploit the hardware performance or generous CPU cache in the case of software. In order to make accurate comparisons between existing hardware or host software and GPGPU software, we included the CRC and mapping as part of the GPGPU algorithm.

Figure 7, Figure 8, and Figure 9 show how the three kernels used by the GPGPU algorithm first decompress the packet (recovering the sigma values that went into the entropy encoder), then apply the predictor, map the data, and finally calculate the CRC. The first kernel depicted in Figure 7 is used to decompress the data and utilizes one device thread for each compressed packet. The more packets passed over in a single buffer transfer, the more threads the device can use to gain parallelism. The next kernel depicted in Figure 8 removes the predictor when in EP decorrelation mode. In EP mode, the packets must be processed in the exact order in which they were compressed because the current frame was subtracted from the previous frame before compression. So, in this second kernel, the threads process the packets in parallel with exactly 4,096 device threads, one for each pixel position. As the predictor is removed from each pixel, the pixel is then mapped accordingly. This will be the mapped packet that will be transferred back to the host. Finally, the third kernel depicted in Figure 9 calculates the CRC. One device thread per packet is used because the CRC calculation, like the decompression calculation, is not

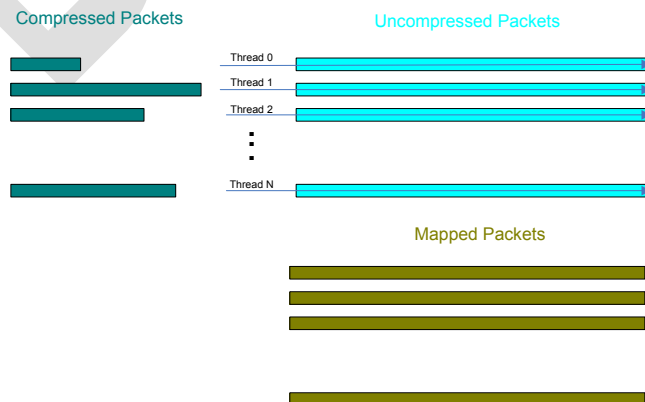
parallelizable within a packet. At the completion of the CRC calculation, the CRC result is appended to the mapped data packet so that the host can verify its correctness.



**Figure 7. Kernel 1 – Decompress Data**



**Figure 8. Kernel 2 – Remove Predictor and Map Data**

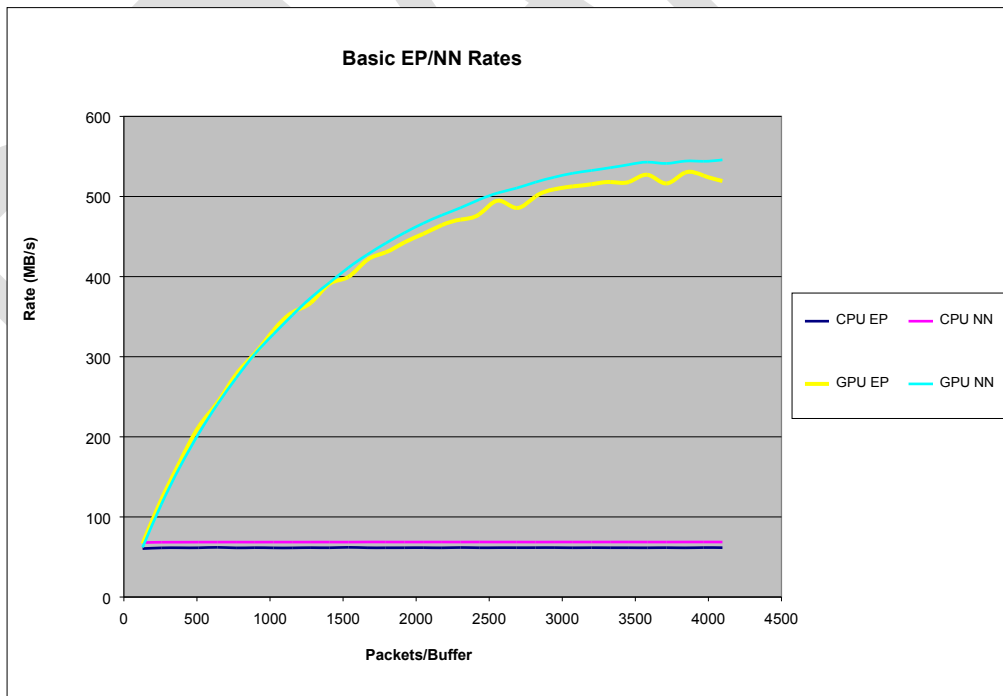


**Figure 9. Kernel 3 – CRC Calculation**

## 4.2. Rice Decompression Software, Central Processing Unit vs. General Purpose Graphics Processing Unit

The Rice Decompression Algorithm was implemented in CUDA and its performance was compared to an existing CPU software implementation. Figure 10 shows the performance results for both the GPGPU and CPU implementation for NN and EP modes. These tests were run using the DL580G5 and S1070 Tesla processing configuration. The compressed data was read from a file into local memory. The data was contrived to have a compression ratio of about 3:1 for both the NN and EP data sets.

First, the CPU software algorithm was used to decompress the data, including the mapping and CRC calculation. The CPU software algorithm was timed for various numbers of packets per buffer. As little as 128 packets to as many as 4,096 packets per buffer were benchmarked. As expected, the number of packets processed per buffer had little effect on the CPU software performance because each packet is always processed one at a time (no parallelism). The rate in millions of bytes per second (MB/s), as shown in the graph in Figure 10, is the rate at which the uncompressed packet data was leaving the algorithm. The input rate of compressed packets is one-third the output rate. The CPU software was able to decompress the data at a rate of approximately 69 MB/s and 62 MB/s for NN and EP modes respectively.



**Figure 10. Software vs. GPGPU Performance**

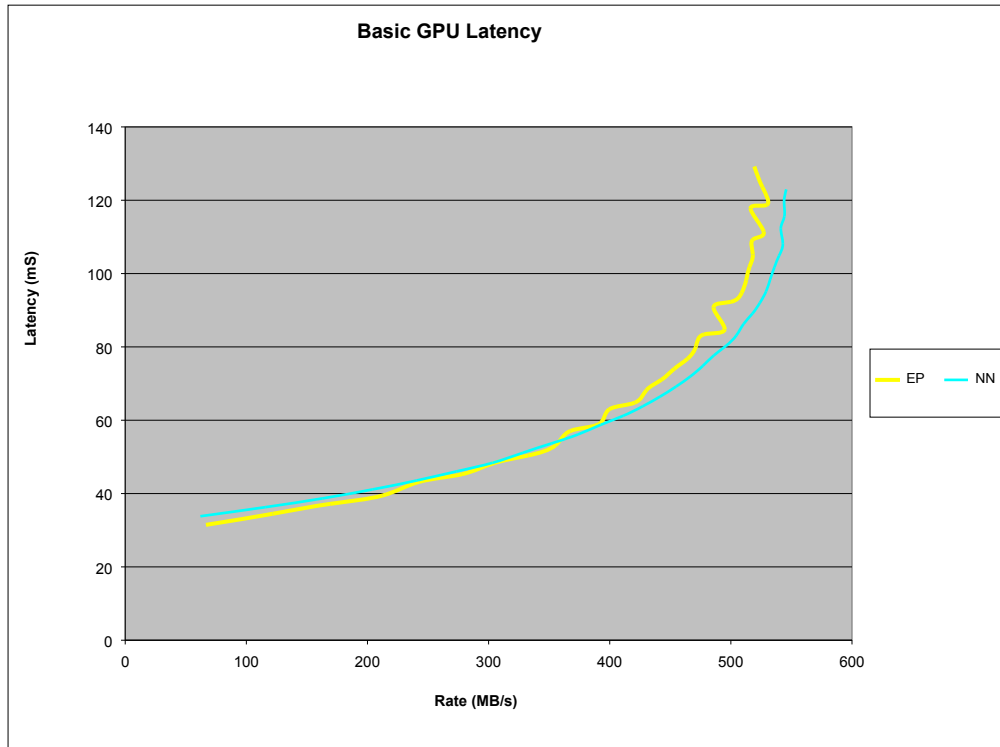
Next, the same benchmark was repeated with the CUDA implementation and GPGPU. In both cases, the host was running a single host thread. The results were timed for the same numbers of packets per buffer (128 – 4,096). The CPU software results were compared to the GPGPU results to verify the correctness of the CUDA code. Additionally, the CRCs were also used to verify

correctness. As expected, the performance of the CUDA algorithm increased rapidly with the number of packets per buffer. This is a direct result of the fact that two of the three kernels (decompression and CRC) used more threads with increasing number of packets per buffer creating improved parallelism. Eventually, the GPGPU becomes fully occupied with threads and no further increases in performance are realized with increasing packet counts. The CUDA implementation achieved 546 MB/s and 530 MB/s for NN and EP modes, respectively, at the higher packet counts.

In any real-time processing system, latency must be a consideration. As previously discussed, in order to exploit the massive number of near simultaneous executing threads available in the GPGPU, data packets must be aggregated (buffered) before passing them to the GPGPU. The act of accumulating packets introduces latency into the processing pipeline. As demonstrated in Figure 10, the larger the number of packets per kernel execution, the faster the packets can be processed. However, for a given data rate, the larger the number of packets accumulated, the greater the latency. In an ideal configuration, the amount of time required to accumulate  $N$  packets would be identical to the amount of time required by the GPGPU to process  $N$  packets. Hence, the minimum latency introduced for this ideal configuration can be easily calculated as twice the amount of time required to accumulate the  $N$  packets required to achieve the desired performance level.

The graph in Figure 11 shows the minimum latency in terms of milliseconds vs. the rate at which the data is being processed as reflected in Figure 10. Latency increases rapidly, as the number of packets per buffer approaches 4,000, corresponding to performance in the area of 500 MB/s. This is due to performance not increasing as rapidly as the number of packets per buffer when the device starts to become fully occupied. Therefore, increasing the number of packets per buffer beyond approximately 4,000 can cause latency to increase without any gain in performance for this device (S1070). The buffer size at which this occurs will depend upon the GPGPU being used, as will be described in Section 4.5.





**Figure 11. Latency**

### **4.3. Timing Analysis of the Basic Compute Unified Device Architecture Rice Decompression Algorithm**

The plots in Figure 12 show a detailed timing analysis of data transfers to and from the device. This figure also shows the time spent in computing by each of the three kernels described in Section 4.1 when decompressing packets in EP mode. The time in seconds represents the time required to process a total of approximately 128,000 packets. The number of packets per buffer was varied as before in the GPGPU performance tests (Figure 10). As displayed in the Timing Analysis graph, the total transfer time of the data remained relatively constant both to and from the device regardless of the size of the buffer being transferred. In other words, transferring many smaller buffers takes about the same amount of time as transferring fewer, larger buffers provided the total number of transferred bytes is the same. This will always be true provided the smallest buffers are large enough to allow efficient DMA transfers. It is notable that the transfer time from the device to host is roughly three times larger than the time it takes to transfer from the host to device. This is due primarily to the 3:1 compression ratio of the data. It is also notable that the data transfers back and forth to the device consume about 45% of the total algorithm time. As demonstrated later in Section 4.4, the transfer time will be hidden by using multiple host thread to overlap transfers and computes.

The time required by the Predictor/Rotate (map) kernel is also relatively constant with increasing packets per buffer. This is because the number of device threads in this kernel is always the same (4,096), regardless of the number of packets per buffer, as previously described in Section 4.1.

Finally, the decompression and CRC kernel show decreasing time as the number of packets per buffer increases resulting from the increased parallelism. The most interesting fact elucidated by this figure is that the total time for the three kernels to perform the compute at high packet counts is very close to the total time required to transfer the data back and forth between the device and host (1060 mS vs. 884 mS). This means that 45% of the time the GPU is idle, waiting for the data transfers to complete. Although CUDA provides a mechanism for overlapping transfers and computes from the same host thread using streams, they cannot guarantee the execution order. This could cause the packets to be processed out of order, which would break the EP model. Another approach is to use multiple host threads. This requires that the overall algorithm maintains a state to ensure that a packet from a given location in the image always gets processed by the same host thread. We address host multithreading algorithms in the next Section 4.4. It is important to note that since the initial testing has been completed, newer versions of CUDA and newer GPGPU architectures have eliminated the requirement that a device context be confined to a single host thread allowing overlapped transfers and computes.

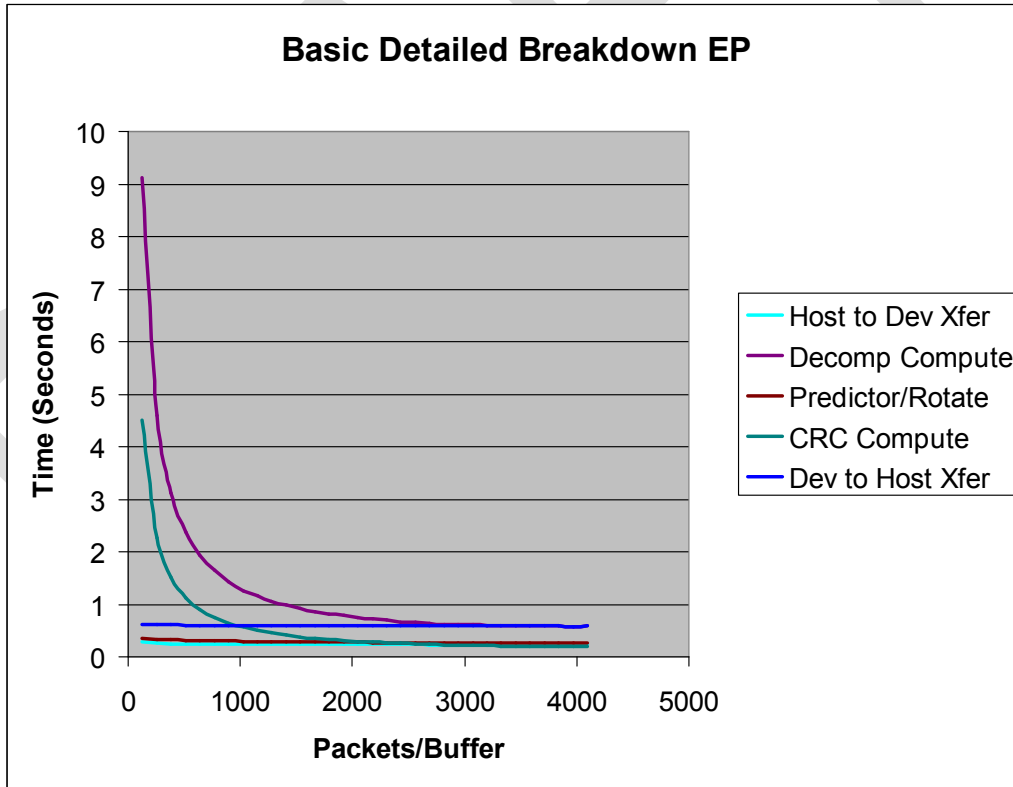


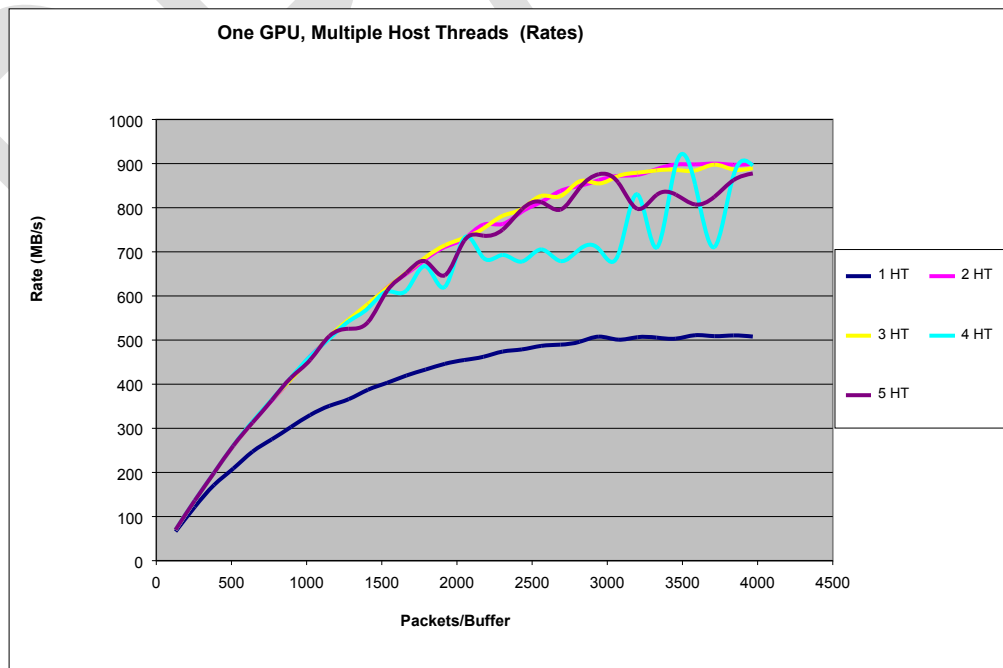
Figure 12. Timing Analysis

#### 4.4. Multiple Host Threads for Rice Decompression

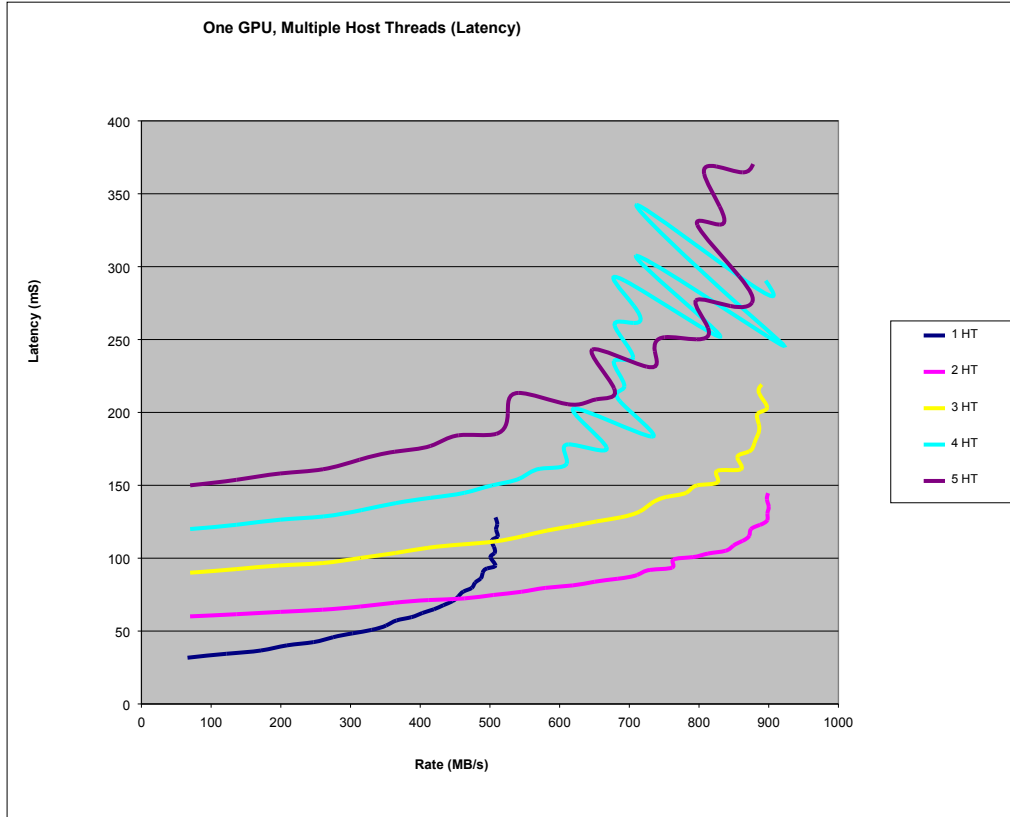
By using more than one host thread, we anticipated overlapping data transfers with computes so that the GPGPU could be fully utilized. While one host thread is waiting for the device to complete the computations, another host thread is waiting for its data to complete the DMA transfer. As mentioned previously, each host thread has its own context on the GPGPU device so

any data sharing between host threads must be done on the host. Because predictor memory is maintained on the device for efficiency, this requires that a packet from a given location on the image must always be processed by the same host thread from one frame to the next.

Figure 13 shows the result for the Tesla S1070 and DL580G5 using one GPGPU device, and anywhere from one to five host threads. These results are for EP mode only. We dropped the NN mode at this point because EP was the more taxing of the two in terms of performance. As Figure 13 illustrates, performance nearly doubles when going from one host thread to two (510 MB/s to 900 MB/s). Based on the results in Figure 12, this is approximately what is expected if the 45% of the time used by Host to Device and Device to Host transfers can be hidden by multiple host threads ( $510 \text{ MB/s} / 0.55 = 927 \text{ MB/s}$ ). Also note that the single-threaded performance for this run is lower than that previously reported in Figure 10 (510 MB/s vs. 530 MB/s). In this particular case, the code has been reconfigured so the Rice decompression is being performed in a stage of a pipeline, as it would be in a real MGS application. Data is being fed to each host thread by a previous stage and the data is then being consumed by another stage at the output. The reduction in performance is due to the overhead associated with filling and emptying the pipeline. As displayed in Figure 13, increasing the number of host threads beyond two does not improve aggregate performance. The GPGPU is fully occupied with two host threads and adding additional threads only increases latency. As seen in Figure 14, a single host thread provides the best latency for a needed rate below 500 MB/s; however, two threads are required to push the performance much beyond 500 MB/s. The next test will focus on using multiple GPGPUs with one or two host threads per GPGPU since the Tesla S1070 contains four GPGPUs.



**Figure 13. Multiple Host Threads (Performance)**



**Figure 14. Multiple Threads (Latency)**

#### **4.5. Multiple Host Threads, Multiple Devices for Rice Decompression**

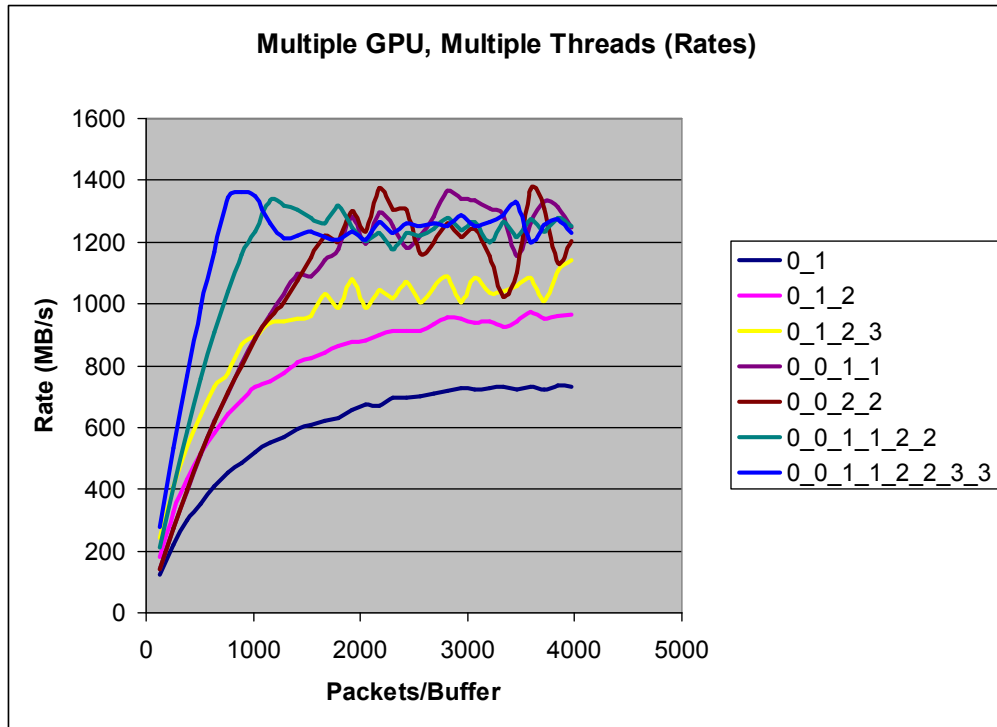
One might incorrectly predict that performance would directly scale with the number of GPGPU devices employed. As described in Section 2, the S1070 uses two HICs to connect the four GPGPUs to the host. Although this provides four times the number of devices for computing, in theory, it would only provide twice the PCIe bandwidth because each of the two GPU pairs must share a single PCIe slot in the host. Hence, one would expect that running two host threads per device on two devices connected through separate HIC cards would at least double the aggregate throughput to 1.8 GB/s (2 x 900 MB/s). In practice, we were not able to get beyond 1.377 GB/s for two, or even more, devices. In an attempt to understand this, we conducted tests to see how both PCIe bandwidth and compute bandwidth scaled with host thread count and device count. The results are shown in Table 2. As can be seen from the table, compute performance scaled directly with the number of GPGPU devices used, whereas DMA bandwidth did not. It appears DMA bandwidth in the aggregate never exceeded what would be expected from a single PCIe x8 Gen 1 slot. One might deduce that the server is using a switch to share the PCIe lanes between the two HIC cards. It could also mean that the Tesla S1070 driver or the server hardware is serializing the data transfers. Using the data in [Table 1](#), we can predict the maximum rate with multiple GPUs for Rice Decompression. Given that the compression ratio is 3:1, 25% of the transfer time is from host to device and 75% is from device to host.

Applying the aggregate data rates from Table 2, the expected maximum transfer rate is 1,395 MB/s,  $(0.25 \times 1607 \text{ MB/s} + 0.75 \times 1948 \text{ MB/s})$ . This assumes that the host-to-device and device-to-host transfers are also serialized. This value agrees very well with the observed result of 1.377 GB/s as seen in Figure 15 which displays two threads and two devices.

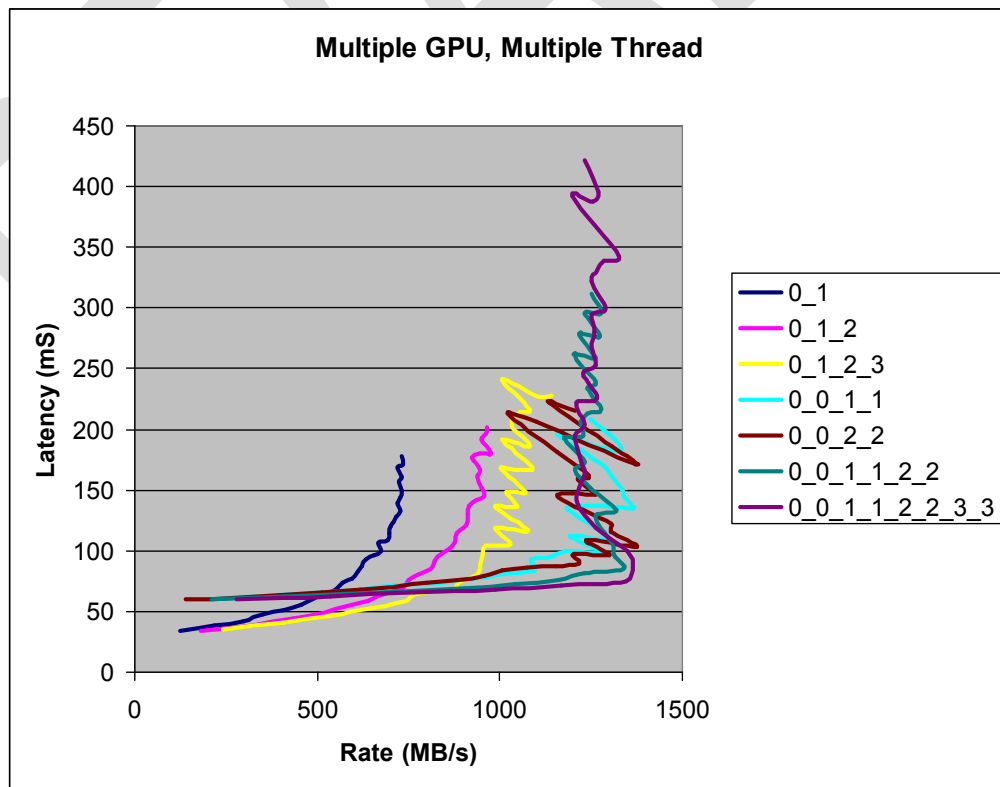
**Table 2. DMA and Compute Bandwidth as a Function of Host Threads and Devices**

		1 Dev, 1Thread	1 Dev, 4 Threads	4 Dev, 4 Threads
Host to Device	Aggregate (MB/s)	1566	1682	1607
	Thread 1	1566	420	416
	Thread 2		410	402
	Thread 3		420	413
	Thread 4		410	402
Device to Host	Aggregate (MB/s)	1556	1962	1948
	Thread 1	1556	494	487
	Thread 2		490	495
	Thread 3		495	500
	Thread 4		498	494
Kernel	Aggregate (MB/s)	33957	34085	135881
	Thread 1	34050	8526	34078
	Thread 2		11344	34065
	Thread 3		11344	34146
	Thread 4		11344	34178

Devices are represented by the numbers 0 through 3 in the legend for Figure 15. For each device the number of threads is indicated by the number of times the device ID is repeated. For example, “0\_0\_2\_2” indicates 2 host threads using device 0 and two host threads using device 2. Likewise, “0\_1\_2\_3” indicates four threads, one thread running on each device, 0-3. Adding additional devices beyond two did not improve performance due to the PCIe bus limitation described earlier. The data in Figure 16 indicates that below 900 MB/s, one host thread running on each of four devices, “0\_1\_2\_3” gave the lowest latency, while above 900 MB/s two host threads on device 0 and two host threads on device 2 gave the lowest latency at “0\_0\_2\_2”.

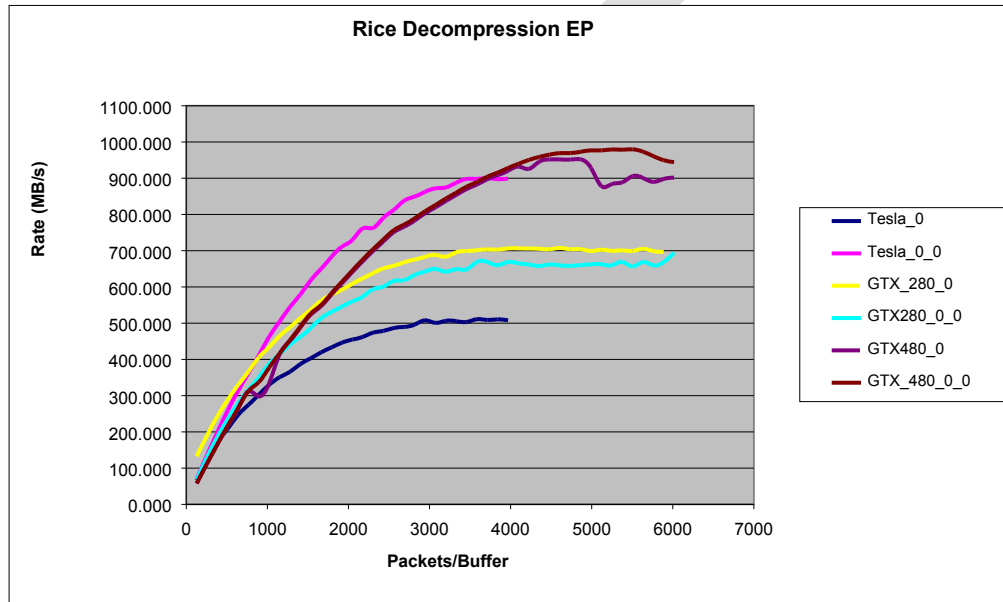


**Figure 15. Multiple Devices and Threads Performance**



**Figure 16. Latency for Multiple Devices and Threads**

Figure 17 shows the results of running the same GPGPU code, but with minor changes to the host code since some changes were required to run on Windows 7 on the two workstations. Only single-device experiments were run because the two workstations only contain one GPGPU each. The GTX 280 uses the same GPGPU chip as the Tesla S1070. The GTX 280 workstation outperformed the Tesla in the single-threaded case (707 MB/s vs. 510 MB/s) because of the improved PCIe x 16 generation 2 bus. Transfer rates between the GPGPU and host on the TAL-2000 workstation nearly doubled the measured transfer rates of the DL580G5 server and S1070.



**Figure 17. Workstation Results**

The GTX 480 workstation performed even better compared to the GTX 280 and S1070 for a single threaded operation (952 MB/s vs. 707 MB/s vs. 510 MB/s, respectively). The TAL-4000 workstation achieved nearly 6 GB/s transfer rates on the PCIe bus and according to NVidia, the GTX 480 has about twice the processing performance of a single S1070 GPGPU device. Interestingly, both the GTX 280 and GTX 480 dropped slightly in performance when using two host threads. This is likely due to a limitation of the GTX drivers not allowing overlap between transfers and computes.

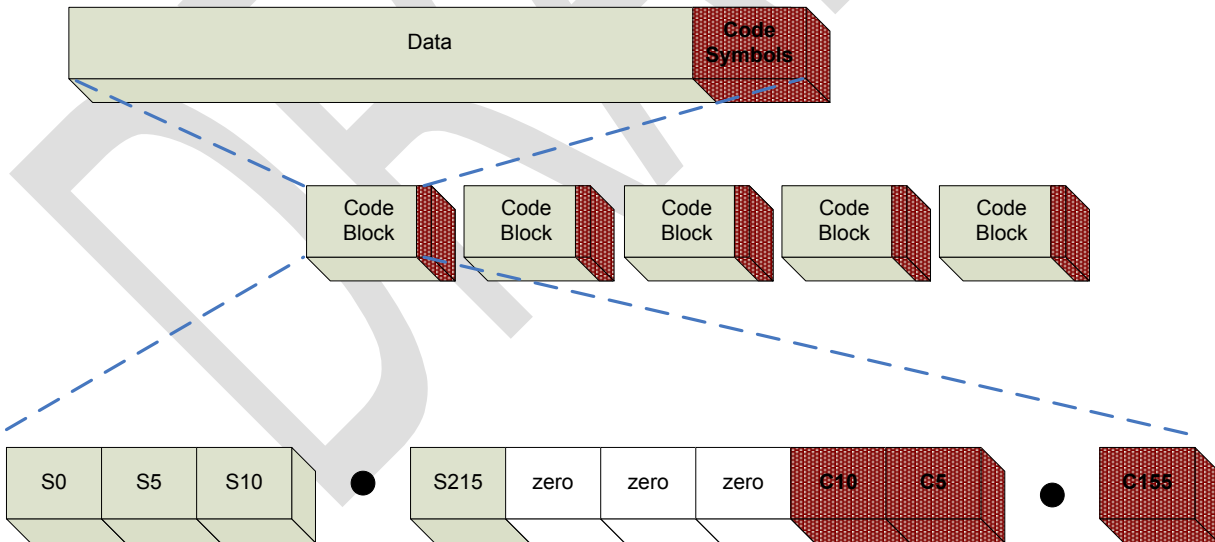
DRAFT



## 5. REED-SOLOMON

Reed-Solomon codes are non-binary, cyclic error correcting codes invented by Irving S. Reed and Gustave Solomon in 1960. They are one of the channel coding techniques recommended by the *CCSDS Green Book 101.0-B-6* [Ref. 2]. A common RS configuration used in space communication systems is the 255/223 code block with 8 bit symbols. This format codes 223 symbols with 32 parity symbols to form a code block of 255 symbols. This configuration allows for the correction of up to 16 symbols per code block. Generally data on a telemetry link will be encapsulated into Transfer Frames (TF). A TF can contain up to 8 code blocks.

To enhance burst error recovery, the code blocks are formed by using every  $n^{th}$  byte, where  $n$  is the number of code blocks in the TF and is commonly referred to as the interleave. If there are not a sufficient number of symbols to complete all of the  $n$  code blocks, virtual fill can be used by padding the data in each block with zeros, as shown in Figure 18. The virtual fill is not transmitted to the ground; it is understood to be there and the ground decoder fills in the zeros before decoding. By using every  $n^{th}$  byte of the message to form the code block data, burst errors tend to be distributed across many code blocks allowing for easier correction of data errors. With an interleave of 8, this configuration can correct a burst error of as much as 1,024 bits (16 symbols/block x 8 bits/symbols x 8 blocks).



**Figure 18. Transfer Frame Format**

For our experiments, we chose an interleave of 8 and a virtual fill of 0, giving us a TF size of 2,040 bytes. To match as closely as possible to a real-life decoder, we also included the four byte-Asynchronous Symbol Marker (ASM) in our data sets. The ASM marks the beginning of every frame. The total frame size is 2,044 bytes, including the ASM. The CPU code we used is based on Phil Karn's Reed-Solomon software, readily available on the internet under GNU General Public License. In our implementation, we pass a large number of TFs at once to the

GPU, similar to what we did for Rice Decompression. The GPU uses 8 device threads per frame to process each of the 8 code blocks. Processing is completed using a single kernel. The kernel first calculates the 32 syndromes for each code block. If all the syndromes equate to zero, then no errors were detected and the process is complete. If one or more of the syndromes is non-zero, then two additional steps must be accomplished – error locating and error calculation. The error location and error calculation are conducted within the same kernel that calculated the syndromes. The final step involves transferring the frame data back to the host along with a status vector that indicates the number of corrections that occurred for each code block, or if a code block was not correctable, i.e., contained more than sixteen errors.

### 5.1. Reed-Solomon Decoding Software vs. General Purpose Graphics Processing Unit

We conducted the RS tests in a manner similar to the Rice Decompression tests. We began with a single host thread and compared the software CPU algorithm with the CUDA algorithm running on the GPGPU. We also started with the same base configuration using the DL580G5 and the Tesla S1070. Each method was timed as a function of the number of frames processed in a single buffer. Each TF frame contains 8 code blocks so the total device threads used per kernel invocation is 8 times the number of TFs. As seen in Figure 19, the device decoding performance increases rapidly with the number of TF frames. In the case of the CPU software, the performance remains independent of the number of TFs, as expected. Each code block in the TF contained the maximum number of errors, which is 16. In the GPGPU tests, data rates peaked at

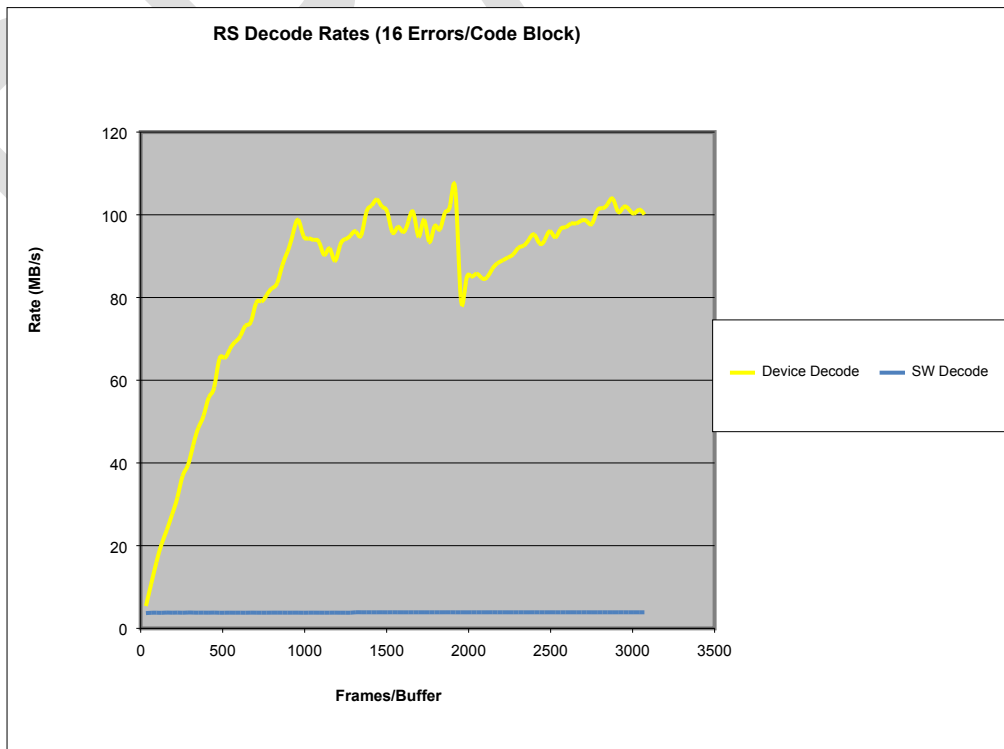
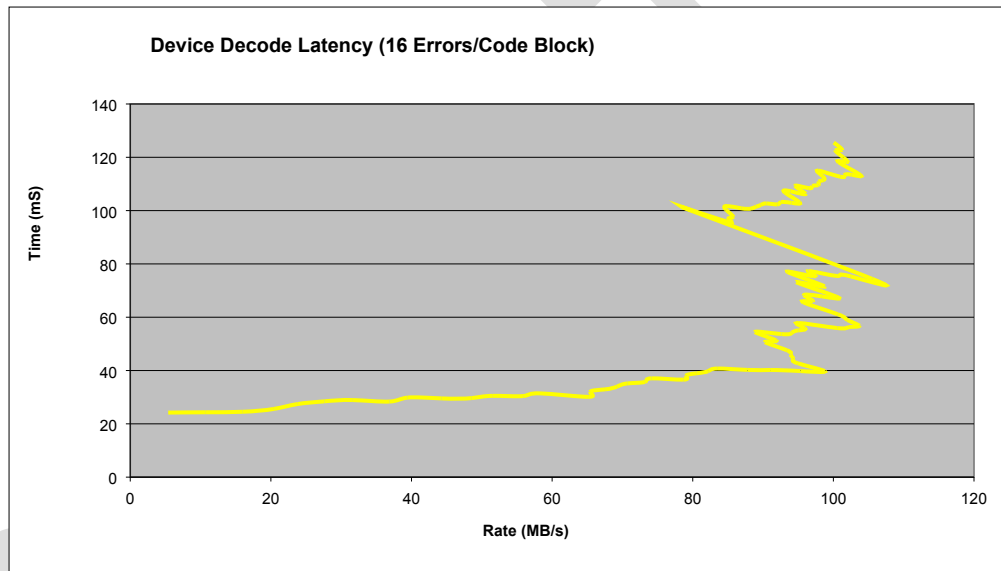


Figure 19. RS Decoding Performance

nearly 107 MB/s, while the CPU tests performance was constant at about 4 MB/s. As with Rice Decompression, latency is an important factor when accumulating data before passing it to the GPGPU. Figure 20 shows the GPU latency, in milliseconds, vs. processing rate in MB/s for GPGPU RS data shown in Figure 19. In the useful range of performance, 4 MB/s to 107 MB/s, the latency is relatively low at 20 to 40 milliseconds. This corresponds from as little as 32 TFs to as many as 1,376 TFs per buffer. In practice, a spacecraft would use multiple links, with each one operating in the tens of MB/s to downlink data. Because there is no interdependence between the TFs on the same link or other links relative to RS decoding, the TFs from all links can be aggregated for processing, making this technique very attractive.



**Figure 20. RS Decoding Latency**

## **5.2. Timing Analysis of the Basic CUDA RS Decoding Algorithm**

Although the Reed Solomon CUDA decoding algorithm only uses a single kernel, it is still useful to look at how compute time for the kernel compares to data transfer time. In this first attempt at using the GPGPU to perform the decoding, we actually copied the entire corrected data buffer back to the host, as well as a status field to indicate the number of symbol corrections or if the frame had any uncorrected errors. Since even under the worst circumstance, no more than 16 out of every 255 symbols could be corrected, an obvious trade would be to only transfer back the symbols that require correction, perhaps using “zero copy.” However, Figure 21 shows that the kernel compute time is many times greater than the time required to transfer the data back and forth to the host. In fact, with 1,376 frames per buffer, the compute requires 6.2 times the time required to complete both transfers. As a result, we determined that pursuing any optimization to transfer only the corrected symbols would not provide significant improvements in overall performance.

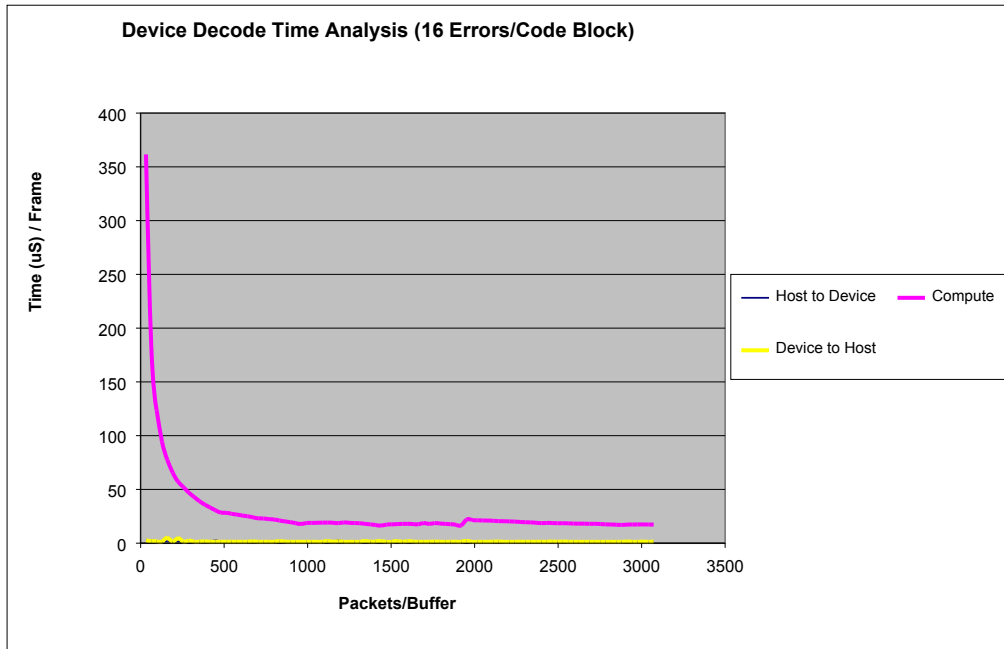


Figure 21. RS Time Analysis

### 5.3. Multiple Host Threads, Multiple Devices for Reed-Solomon Decoding

Based on the results from Section 5.2, RS is compute bound and should not benefit significantly from overlapped I/O; however, it should scale well with increasing device counts. Figure 22 validates this premise.

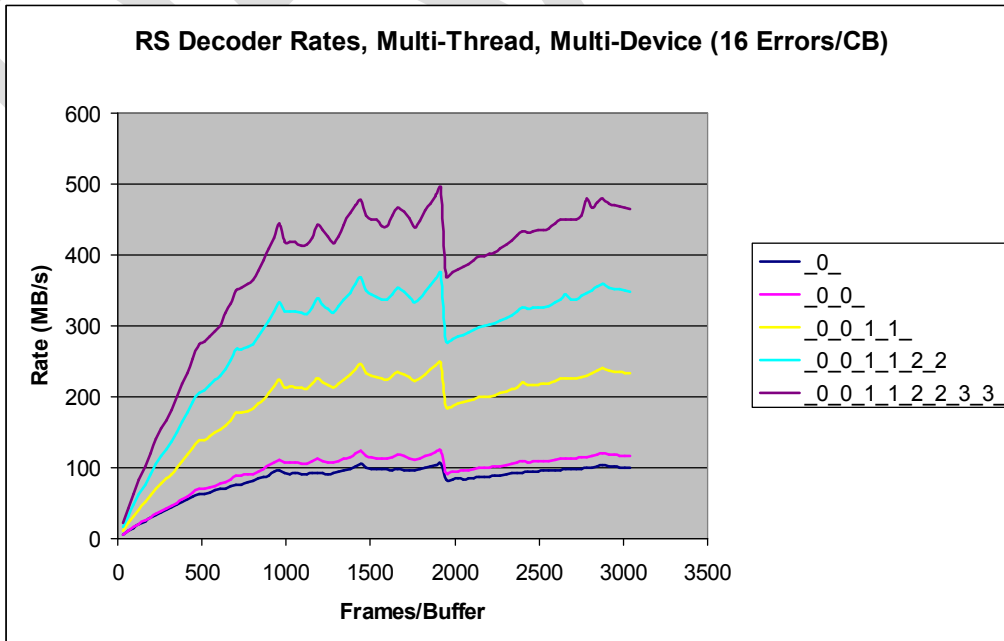
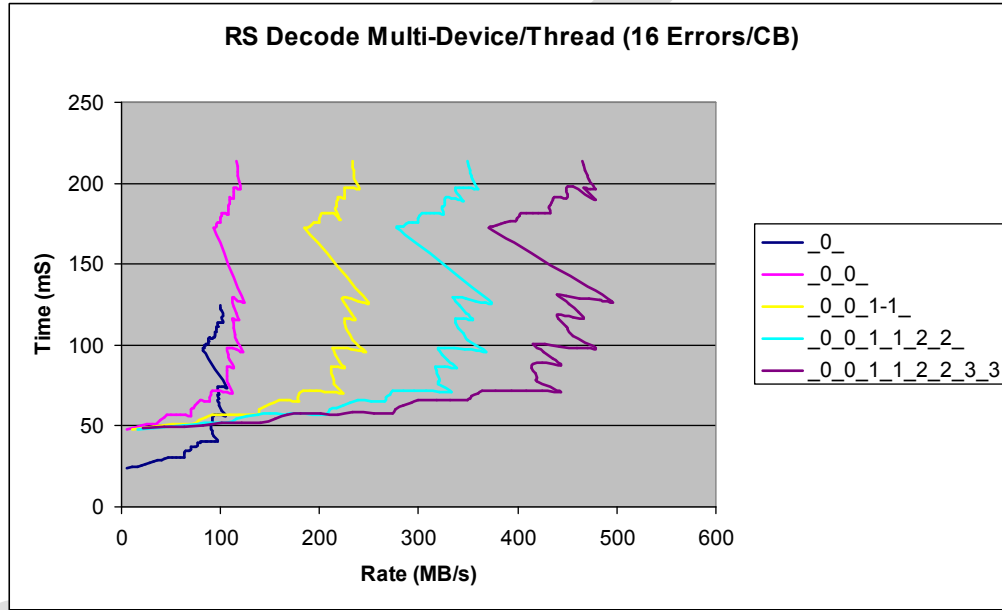


Figure 22. Multiple Threads, Multiple Devices RS Decoding

For the multi-threaded RS benchmarks, the software was restructured to make it more like a stage in a pipeline, just as we did for the Rice Decompression multi-threaded code. As shown in Figure 22, when going from one thread on device 0, “\_0\_”, to two host threads on device 0, “\_0\_0\_”, it made little difference in performance – 125 MB/s vs. 107 MB/s. As also predicted, increasing the number of devices causes the performance to increase proportionally. Furthermore, in Figure 23, it is clear that increasing from one host thread to two host threads doubles the latency; whereas, adding devices does not impact latency hardly at all.

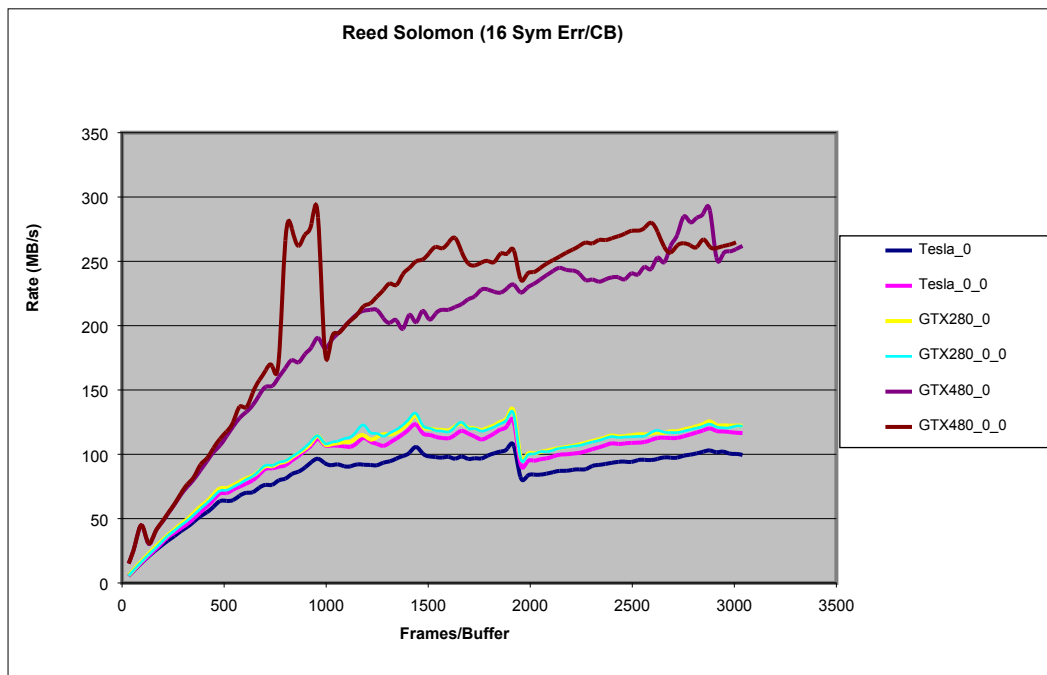


**Figure 23. Multiple Host Thread, Multiple Device RS Latency**

The case where one host thread is run on each device is not shown in the figures, but was benchmarked. In that instance, performance peaks at 350 MB/s, as opposed to 494 MB/s when two host threads are used on each device. It appears that multi-threading of multiple devices can offer about a 41% improvement in performance as opposed to only a 16% improvement when using a single multi-threaded device. It is not clear at this point, why multiple host threads provide such an improvement in performance when using multiple devices.

Finally, we looked at using two different workstations to run the same CUDA GPGPU code, making slight changes to the host code to allow it to run under Windows 7. As displayed in Figure 24, and as expected, the GTX 280 performed very similarly to the two host thread, single device result from the DL580G5 and Tesla S1070 combination. The GTX 480 workstation, on the other hand, was able to reach 289 MB/s with a single thread, more than double the 125 MB/s on the Tesla S1070 using two threads and one device. Again, this is primarily due to the doubling in performance of NVidia's Fermi architecture over the previous Tesla architecture. Also noteworthy is the peak in performance at 800 through 960 frames per buffer in the GTX 480 workstation, two host threads, benchmark, “GTX480\_0\_0”. The GTX 480 has the ability to

run up to 16 kernels simultaneously if they fit in the device. It appears that at this “sweet spot,” the two host threads are sharing the GPGPU, running both of their kernels simultaneously.



**Figure 24. Workstation Performance**

## 6. CONCLUSION

Our results show that the GPGPU, in the case of Rice Decompression and Reed Solomon decoding, has considerable potential for performing satellite communication Data Signal Processing. The GPGPU showed significant performance increases over traditional techniques with minimal increases in latency. We demonstrated that even commodity graphics cards like the GTX 280 and GTX 480, running on relatively low-end workstations, were capable of outperforming traditional firmware and software based solutions. In addition, GPGPU solutions can offer 20-times to 100-times reduction in cost depending on the GPU solution selected. For example, a 50 MB/s firmware RS solution can cost \$50K, compared to a \$2.3K C2050, single GPU Tesla processing or a \$500 GTX 480 solution that can both provide in excess of 100 MB/s.

We demonstrated that when designing a GPGPU based system, it is very important to consider I/O bandwidth, especially in the case of Rice Decompression. The PCIe bus DMA bandwidth on the DL580G5 was shown to be the limiting performance factor when implementing the Rice algorithm on multiple GPGPUs. Not considered in our experiments was the added burden of additional I/O from other devices in the host. There obviously needs to be some I/O associated with bringing the data into the host and passing results on to other Data Processing Subsystems. Because the memory bus, FSB, and QPI are all involved in I/O, it may be that one of them would be the bottleneck in a real Data Acquisition Subsystem (DAS) front end as opposed to the GPGPU.

Although we were able to demonstrate significant performance improvement using multiple GPGPUs, in all cases, a single GPGPU was able to exceed our current data rate requirements by a factor of at least two. It is unlikely that satellite downlink requirements will grow at a faster pace than GPGPU technology. Therefore, it may be practical to assign multiple functions to the same GPGPU. For example, one GPGPU may be able to process both the Reed Solomon Decoding and the Rice Decompression. It may also be useful to share a four-GPGPU Tesla processing between two hosts, one running the DAS front end and the second running parts of a second stage image processing system. For lower throughput requirements, it may be useful to consider lower-end versions of the GPGPU, either in terms of commodity GeForce (GTX) cards or Nvidia's Quadro FX series cards.

DRAFT



## 7. REFERENCES

1. *Lossless Data Compression. Recommendation for Space Data System Standards, CCSDS 121.0-B-2. Blue Book.* Issue 2. Washington, D.C.: CCSDS, May 2012.
2. *Telemetry Channel Coding. Recommendation for Space Data System Standards, CCSDS 101.0-B-6. Blue Book.* Issue 6. Washington, D.C.: CCSDS, October 2002.

## DISTRIBUTION

1	MS0576	Feddema, John T	05521
2	MS0567	Loughry, Thomas	05531
1	MS0899	Technical Library	09536 (Electronic Copy)

DRAFT

DRAFT

DRAFT

DRAFT



**Sandia National Laboratories**